Lecture #13: Query Planning & Optimization I

15-445/645 Database Systems (Fall 2021)
https://15445.courses.cs.cmu.edu/fall2021/
Carnegie Mellon University
Lin Ma

1 Overview

Because SQL is declarative, the query only tells the DBMS what to compute, but not how to compute it. Thus, the DBMS needs to translate a SQL statement into an executable query plan. But there are different ways to execute each operator in a query plan (e.g., join algorithms) and there will be differences in performance among these plans. The job of the DBMS's optimizer is to pick an optimal plan for any given query.

The first implementation of a query optimizer was IBM System R and was designed in the 1970s. Prior to this, people did not believe that a DBMS could ever construct a query plan better than a human. Many concepts and design decisions from the System R optimizer are still in use today.

There are two high-level strategies for query optimization.

The first approach is to use static rules, or *heuristics*. Heuristics match portions of the query with known patterns to assemble a plan. These rules transform the query to remove inefficiencies. Although these rules may require consultation of the catalog to understand the structure of the data, they never need to examine the data itself.

An alternative approach is to use *cost-based search* to read the data and estimate the cost of executing equivalent plans. The cost model chooses the plan with the lowest cost.

Query optimization is the most difficult part of building a DBMS. Some systems have attempted to apply machine learning to improve the accuracy and efficiency of optimizers, but no major DBMS currently deploys an optimizer based on this technique.

Logical vs. Physical Plans

The optimizer generates a mapping of a *logical algebra expression* to the optimal equivalent physical algebra expression. The logical plan is roughly equivalent to the relational algebra expressions in the query.

Physical operators define a specific execution strategy using an access path for the different operators in the query plan. Physical plans may depend on the physical format of the data that is processed (i.e. sorting, compression).

There does not always exist a one-to-one mapping from logical to physical plans.

2 Relational Algebra Equivalence

Much of query optimization relies on the underlying concept that the high level properties of relational algebra are preserved across equivalent expressions. Two relational algebra expressions are *equivalent* if they generate the same set of tuples.

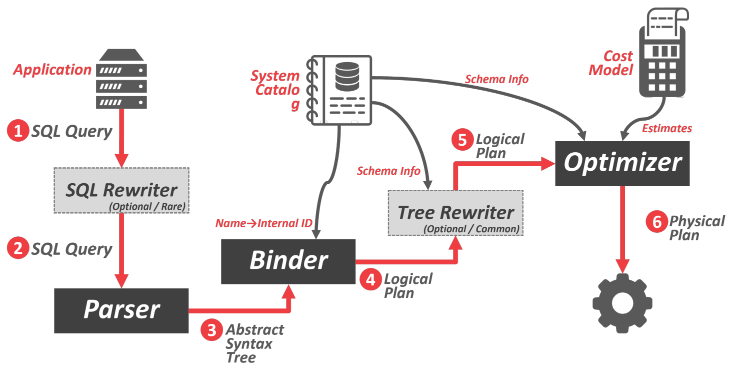


Figure 1: Architecture Overview – The application connected to the database system and sends a SQL query, which may be rewritten to a different format. The SQL string is parsed into tokens that make up the syntax tree. The binder converts named objects in the syntax tree to internal identifiers by consulting the system catalog. The binder emits a logical plan which may be fed to a tree rewriter for additional schema info. The logical plan is given to the optimizer which selects the most efficient procedure to execute the plan.

This technique of transforming the underlying relational algebra representation of a logical plan is known as *query rewriting*.

One example of relational algebra equivalence is *predicate pushdown*, in which a predicate is applied in a different position of the sequence to avoid unnecessary work. Figure 2 shows an example of predicate pushdown.

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

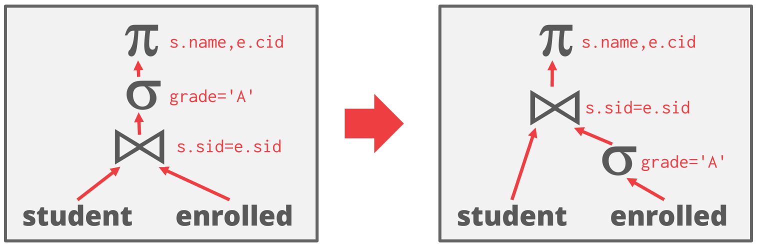


Figure 2: Predicate Pushdown: – Instead of performing the filter after the join, the filter can be applied earlier in order to pass fewer elements into the filter.

3 Logical Query Optimization

Some selection optimizations include:

- Perform filters as early as possible (predicate pushdown).
- Reorder predicates so that the DBMS applies the most selective one first.
- Breakup a complex predicate and pushing it down (split conjunctive predicates).

An example of predicate pushdown is shown in Figure 2.

Some projection optimizations include:

- Perform projections as early as possible to create smaller tuples and reduce intermediate results *(projection pushdown)*.
- Project out all attributes except the ones requested or requires.

An example of projection pushdown in shown in Figure 3.

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
   AND e.grade = 'A'
```

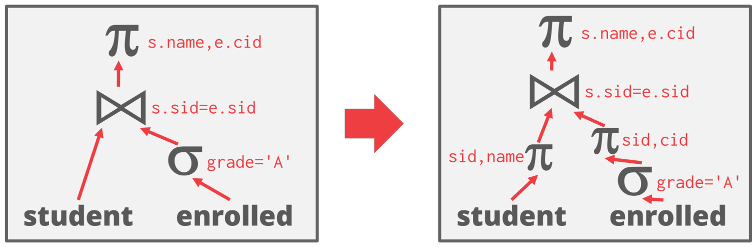


Figure 3: Projection Pushdown – Since the query only asks for the student name and ID, the DBMS can remove all columns except for those two before applying the join.

Another optimization that a DBMS can use is to remove impossible or *unnecessary predicates*. In this optimization, the DBMS elides evaluation of predicates whose result does not change per tuple in a table. Bypassing these predicates reduces computation cost. Figure 4 shows two examples of unnecessary predicates.

```
SELECT * FROM A WHERE 1 = 0; ✗
```

```
SELECT * FROM A;
```

Figure 4: Unnecessary Predicates – The predicate in the first query will always be false and can be disregarded. The former query can be rewritten as the latter query to produce the same result but save on computation.

A similar optimization is *merging predicates*. An example of this optimization is shown in Figure 5.

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150;
```

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 150;
```

Figure 5: Merging Predicates – The WHERE predicate in query 1 has redundancy as what it is searching for is any value between 1 and 150. Query 2 shows the more succinct way to express request in query 1.

The ordering of JOIN operations is a key determinant of query performance. Exhaustive enumeration of all possible join orders is inefficient, so join-ordering optimization requires a cost model. However, we can still eliminate *unnecessary joins* with a heuristic approach to optimization. An example of join elimination is shown in Figure 6.

```
SELECT A1.*
  FROM A AS A1 JOIN A AS A2
    ON A1.id = A2.id;
```

```
SELECT * FROM A;
```

Figure 6: Join Elimination – The join in query 1 is wasteful because every tuple in A must exist in A. Query 1 can instead be written as query 2.

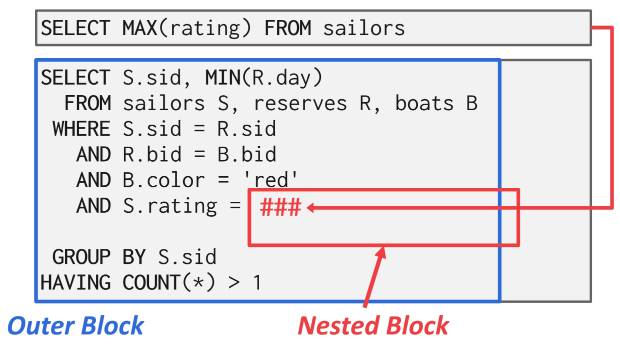
The DBMS can also optimize nested sub-queries without referencing a cost model. There are two different approaches to this type of optimization:

- Re-write the query by de-correlating and / or flattening it. An example of this is shown in Figure 7.
- Decompose the nested query and store the result to a temporary table. An example of this is shown in Figure 8.

```
SELECT name FROM sailors AS S
 WHERE EXISTS (
    SELECT * FROM reserves AS R
     WHERE S.sid = R.sid
       AND R.day = '2018-10-15'
 )
```

```
SELECT name
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = '2018-10-15'
```

Figure 7: Subquery Optimization - Rewriting The former query can be rewritten as the latter query by rewriting the subquery as a JOIN. Removing a level of nesting in this way effectively *flattens* the query.



```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Figure 8: Subquery Optimization - Decomposition – For complex queries with subqueries, the DBMS optimizer may break up the original query into blocks and focus on optimizing each individual block at a a time. In this example, the optimizer decomposes a query with a nested aggregation by pulling the nested query out into its own query, and subsequently using this result to realize the logic of the original query.